组成原理实验课程第_6_次实报告

实验名称	单周期 CPU 实现			班级	李涛
学生姓名	艾明旭	学号	2111033	指导老师	董前琨
实验地点	A306		实验时间	5.30/6.6	

1、 实验目的

1. 理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。

2. 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。

3. 熟悉并掌握单周期 CPU 的原理和设计。

4. 进一步加强运用 verilog 语言进行电路设计的能力。

5. 为后续设计多周期 cpu 的实验打下基础。

2、 实验内容说明

预习:

1) 熟知 MIPS 指令类型，深入理解常用指令的功能和编码;

2) 归纳常用的 MIPS 指令，确定自己准备实现的 MIPS 指令;

根据实验指导书实验六相关部分完成单周期 CPU 设计实验，在原始实验基础上进行改进，按照如下要求完成实验报告：

1、原始代码实验验证使用实验箱验证，可以不进行仿真，验证时在运行一系列指令之后，实验箱拍照，对比说明各个寄存器中的数据是否是执行正确的结果即可。

1) 确认单周期 CPU 的设计框图的正确性;

2) 编写 verilog 代码，将汇编程序翻译为二进制，内嵌到指令 ROM 中;

3) 对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材 料;

4) 完成调用单周期 CPU 的外围模块的设计，并编写代码;

5) 对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。

2、改进要求，针对目前 CPU 可运行 R 型和 I 型 MIPS 指令，各补充一条新的指令，需要修改的 ALU 模块可参照实验四当时的 ALU 改进。改进时注意以下几点：

1）MIPS 指令格式要使用规范格式；

2）指令执行验证需要修改 inst_rom 中预存储的 16 进制指令数据；

3）注意代码中单周期 CPU 模块（

single_cycle_cpu)中实现主要功能使用的都是组合逻辑，

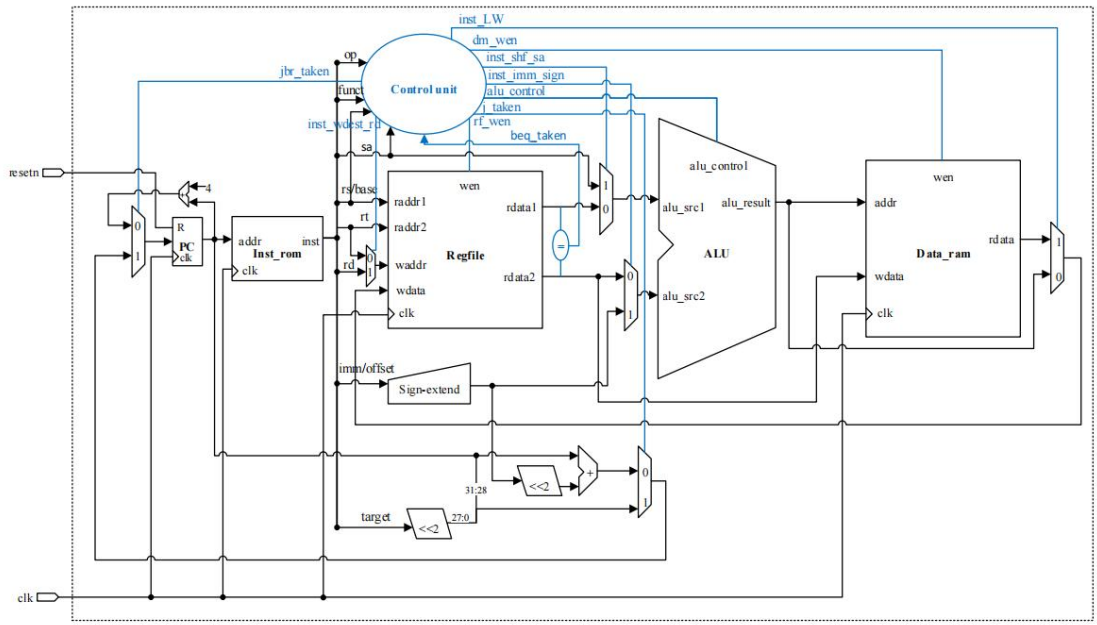
改进过程中避免使用 alway(clk)这样的时序逻辑。

3、实验原理图使用实验指导书的图 7.3 即可，无需修改。

4、实验报告中要有介绍分析的内容，针对实验箱照片，要解释图中信息，是否验证成功。

3、 实验原理图

1.单周期 CPU 的实现框图如下：



2.单周期 CPU 参考设计的顶层模块框图如下：

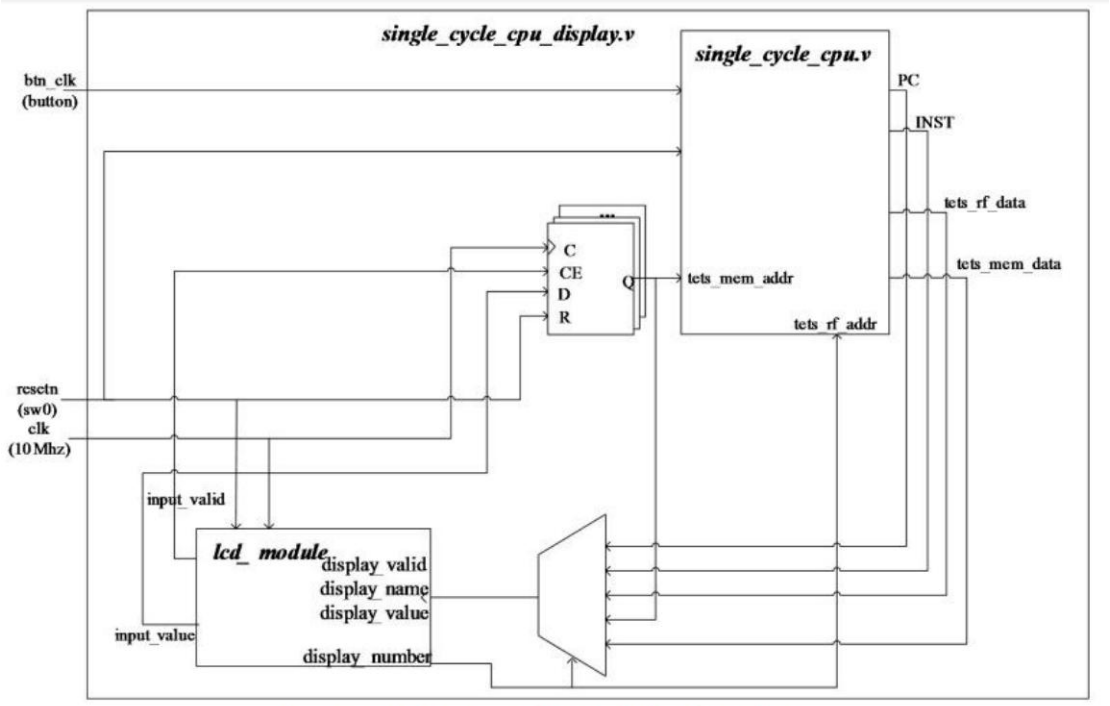


图 7.4 单周期 CPU 参考设计的顶层模块框图

4、 实验步骤

1.使用源码对初始结果进行复现并且验证，验证结果详见四.实验结果分析。

2.对于 R 型和 I 型指令，各补充一条新的指令，并根据之前所实现的 ALU，在第四次实验中，我添加了**非运算、与非运算和大于置位。这里将其修改——**

（1）大于置位（SGT-set greater than）——设计为 R 型。

（2）与非运算（NAND）——设计为 I 型。

3.具体设计：

（1）I 型指令——与非运算（NAND）：

op	rs	rt	immediate
100 001	00000	11110	0000 0000 0000 0001

对应 32 位二进制为：1000 0100 0001 1110 0000 0000 0000 0001

转换为 16 进制为：841E 0001

对应汇编指令为——

xori $30, $0, 1

①先将 op 的值 100 001;

②对于 rs 寄存器，根据 I 型指令的要求，将后 16 位进行移位。为了能够使与非运算结果更明显，我选择 REG0 寄存器，因为其数值一直显示为 0;

③将移动后的数字存入 rt 寄存器，选择 REG1E 即 11110 对应的寄存器来存储计算结果。

④令立即数设为 1 即（0000 0000 0000 0001）。同时由于只需要 5 个二进制位就足够，因此将后十六位数字设计固定为 1，最后存入指令 inst_rom 中。

（2）R 型指令——大于置位（SGT）：

op	rs	rt	rd	shamt	funct
000 000	00011	00010	11111	00000	000 000

对应 32 位二进制为：0000 0000 0110 0010 1111 1000 0000 0000

转换为 16 进制为：0062 F800

对应汇编指令为：

llt $31, $3, $2

①先将 op 的值设为 000 000;

②令 rs=00011 即 3 号寄存器，rt=00010 即 2 号寄存器，rd=11111 即 1F 号寄存器。

③shamt 字段没有使用的必要，并且将 funct 的值设为全 0。

4.代码修改：

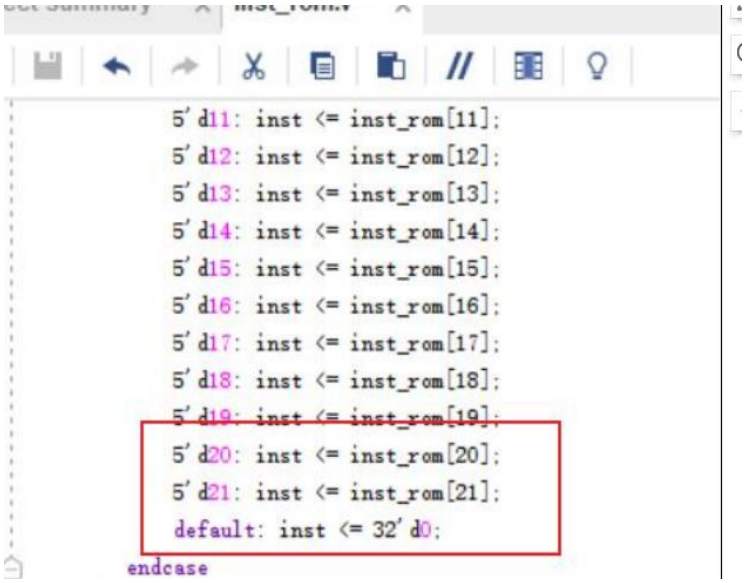
（1）inst_rom.v：

①为 inst_rom 增加两条指令即 inst_rom[19]与 inst_rom[20]，同时令 inst_rom[21]变为之前 inst_rom[19]位置对应的控制跳转重新开始的 j 00H 指令。

```
assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2   | 跳转到指令34H
assign inst_rom[12] = 32'h24010004; // 30H: addiu $1 ,$0,#4   | 不执行
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1) | $10 = 0000_000DH
assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3   | 不跳转
assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1   | $11 = 0000_0000H
assign inst_rom[16] = 32'hAC0B001C; // 40H: sw    $11,#28($0) | Men[0000_001CH] = (
assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0) | Mem[0000_0010H] = (
assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12     | [R12] = 000C_0000H
assign inst_rom[19] = 32'h841E0001; //与非(NAND) xori $30, $0, 1
assign inst_rom[20] = 32'h0062F800; // 大于置位 (SGT) llt $31, $3, $2
assign inst_rom[21] = 32'h08000000; // j 00H | 跳转指令00H

//读指令,取4字节
always @(*)
begin
```

②补充对应 inst_rom[20]与 inst_rom[21]的输出，注意此时 inst_rom[19]的输出无需更改。即增加 5'd20 与 5'd21。



```
        5'd11: inst <= inst_rom[11];
        5'd12: inst <= inst_rom[12];
        5'd13: inst <= inst_rom[13];
        5'd14: inst <= inst_rom[14];
        5'd15: inst <= inst_rom[15];
        5'd16: inst <= inst_rom[16];
        5'd17: inst <= inst_rom[17];
        5'd18: inst <= inst_rom[18];
        5'd19: inst <= inst_rom[19];
        5'd20: inst <= inst_rom[20];
        5'd21: inst <= inst_rom[21];
        default: inst <= 32'd0;
    endcase
```

（2）single_cycle_cpu.v：

①添加新的指令：增加 wire inst_NAND 与 wire inst_SGD。并且 assign 对应的 inst 指令。

```
// 实现指令列表
wire inst_ADDU, inst_SUBU , inst_SLT, inst_AND;
wire inst_NOR , inst_OR   , inst_XOR, inst_SLL;
wire inst_SRL , inst_ADDIU, inst_BEQ, inst_BNE;
wire inst_LW  , inst_SW   , inst_LUI, inst_J;
wire inst_NAND, inst_SGT;

assign inst_NAND = (op == 6'b100001);
assign inst_SGT = op_zero & sa_zero & (funct == 6'b000000);

assign inst_ADDU  = op_zero & sa_zero    & (funct == 6'b100001);// 无符号加法
assign inst_SUBU  = op_zero & sa_zero    & (funct == 6'b100011);// 无符号减法
assign inst_SLT   = op_zero & sa_zero    & (funct == 6'b101010);// 小于则置位
assign inst_AND   = op_zero & sa_zero    & (funct == 6'b100100);// 逻辑与运算
assign inst_NOR   = op_zero & sa_zero    & (funct == 6'b100111);// 逻辑或非运算
```

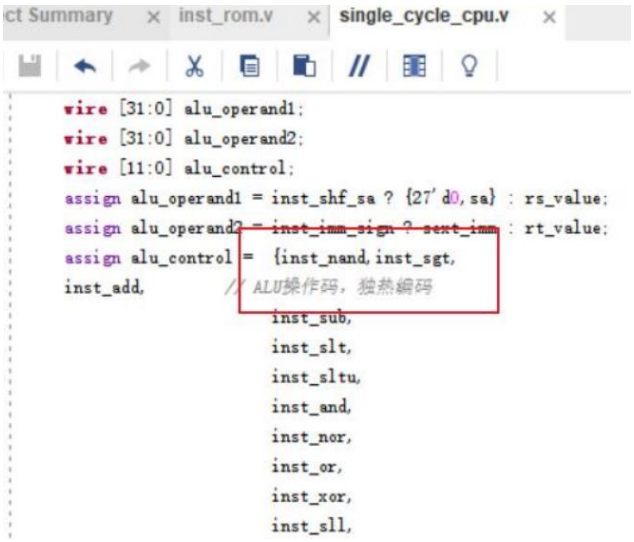
②添加控制信号，需要让 ALU 进行接收：

```
// 传递到执行模块的ALU源操作数和操作码
wire inst_add, inst_sub, inst_slt,inst_sltu;
wire inst_and, inst_nor, inst_or, inst_xor;
wire inst_sll, inst_srl, inst_sra,inst_lui;

wire inst_nand, inst_sgt;
assign inst_nand = inst_NAND;
assign inst_sgt = inst_SGT;

assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
assign inst_sub = inst_SUBU; // 减法
assign inst_slt = inst_SLT;  // 小于置位
assign inst_sltu= 1'b0;      // 暂未实现
```

③对 alu_control 中的独热码进行补充,添加对应的独热编码：



④添加写存信号给新增的指令:R 型 SGT 需要写回到 rd；I 型指令 NAND 异或需要写回 rt。

```
//------------------------------{写回}begin------------------------------//
    wire inst_wdest_rt;    // 寄存器堆写入地址为rt的指令
    wire inst_wdest_rd;    // 寄存器堆写入地址为rd的指令
    assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI| inst_NAND ;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
                         | inst_OR   | inst_XOR  | inst_SLL | inst_SRL | inst_SGT;
;
    // 寄存器堆写使能信号，非复位状态下有效
    assign rf_wen   = (inst_wdest_rt | inst_wdest_rd) & resetn;
    assign rf_waddr = inst_wdest_rd ? rd : rt;          // 寄存器堆写地址rd或rt
    assign rf_wdata = inst_LW ? dm_rdata : alu_result;// 写回结果，为load结果或ALU结果
```

⑤对 I 型异或操作的立即数进行调整，即对后 16 位进行 32 位拓展，实际上就是对于我存进去的立即数 1 进行拓展。

```
assign inst_sra = 1'b0;       // 暂未实现
assign inst_lui = inst_LUI;  // 立即数装载高位

wire [31:0] sext_imm;
wire   inst_shf_sa;    //使用sa域作为偏移量的指令
wire   inst_imm_sign;  //对立即数作符号扩展的指令
assign sext_imm      = {{16{imm[15]}}, imm};// 立即数符号扩展
assign inst_shf_sa   = inst_SLL | inst_SRL;
assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW| inst_NAND;

wire [31:0] alu_operand1;
```

(3) alu.v：

①将 alu_control 从 4 位再改回 14 位，即[13:0]。

```
module alu(
    input  [13:0] alu_control,  // ALU控制信号,重新改为14位
    input  [31:0] alu_src1,     // ALU操作数1,为补码
    input  [31:0] alu_src2,     // ALU操作数2,为补码
    output [31:0] alu_result    // ALU结果
    );

    // ALU控制信号，独热码->四位二进制
    wire alu_add;   //加法操作
```

②修改对应的 alu_control 的对应数字：根据之前在 single_cycle_cpu.v 中对 alu_control 独热码的编码顺序，先加入 alu_sgt，再加入 alu_nand。

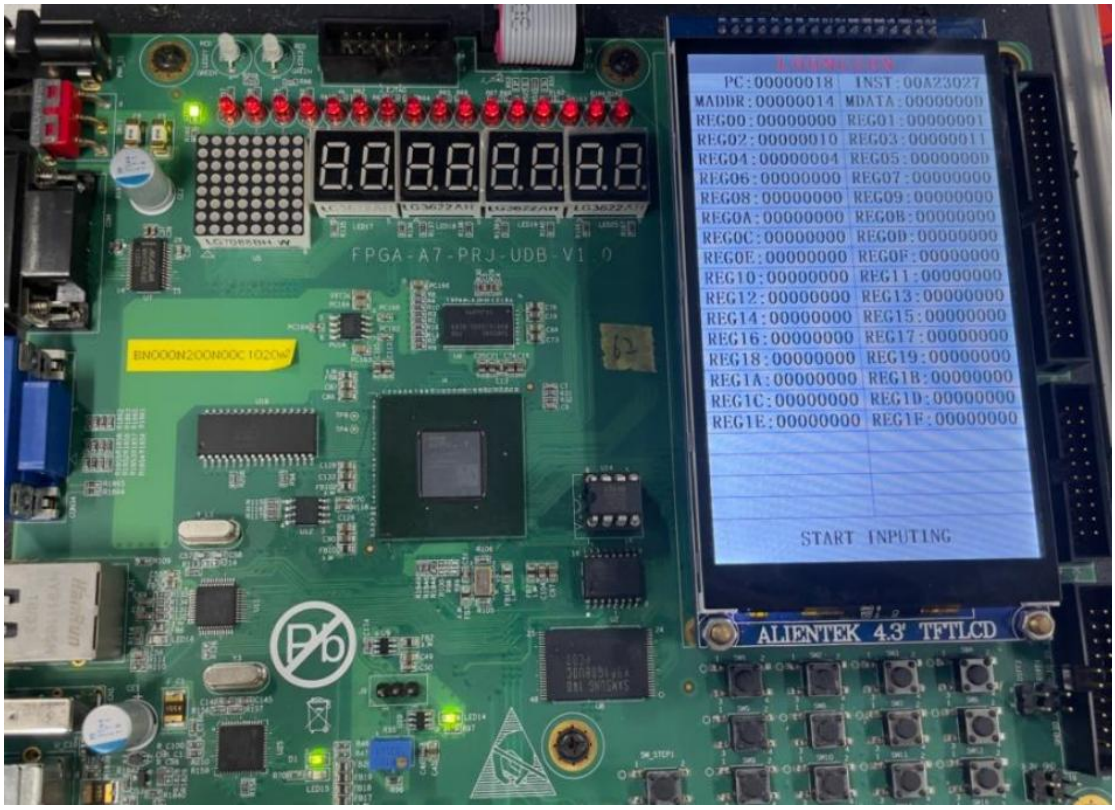
```
    //assign alu_not  = alu_control==4'b1101;
    assign alu_sgt = alu_control==4'b1101;
    assign alu_nand  = alu_control==4'b1110;
```

（分布介绍依次完成了哪些代码修改，从而实现了什么样的功能）

5、 实验结果分析

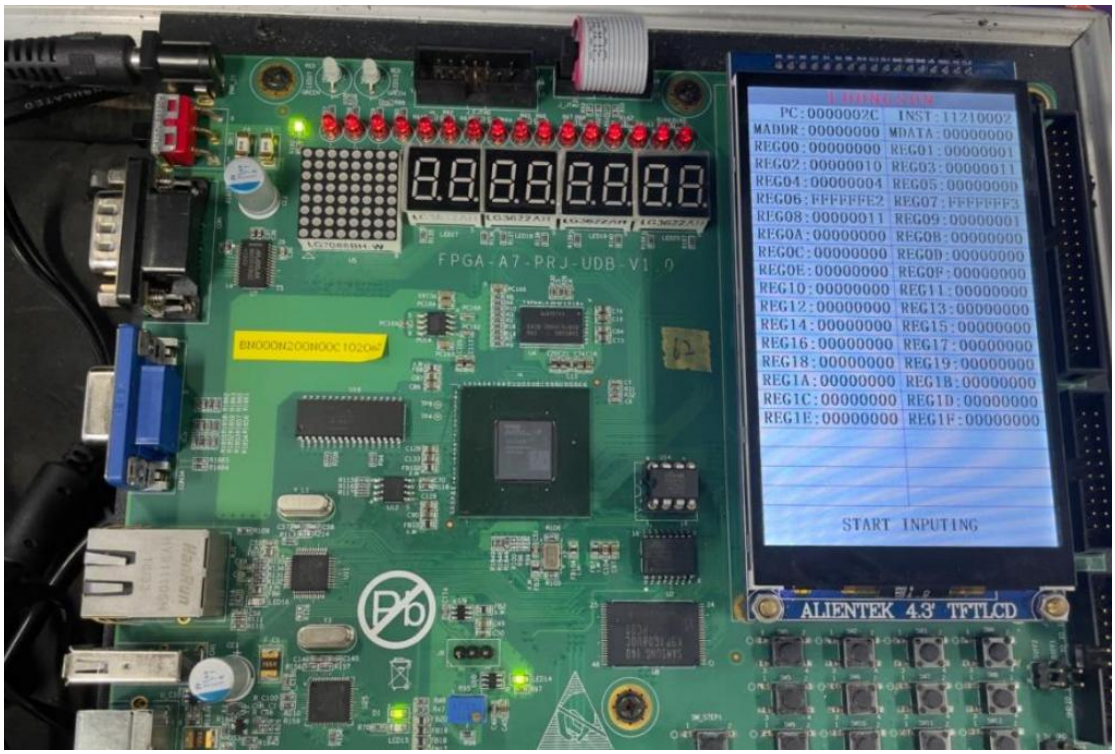
1. 对源码进行验证：（第一次实验）

（1）运行至 PC=18 时：



运行前六条指令得到如上结果，01~05 寄存器的值均与预期一致，查看 0x00000014 内存地址得到的值也与预期一致

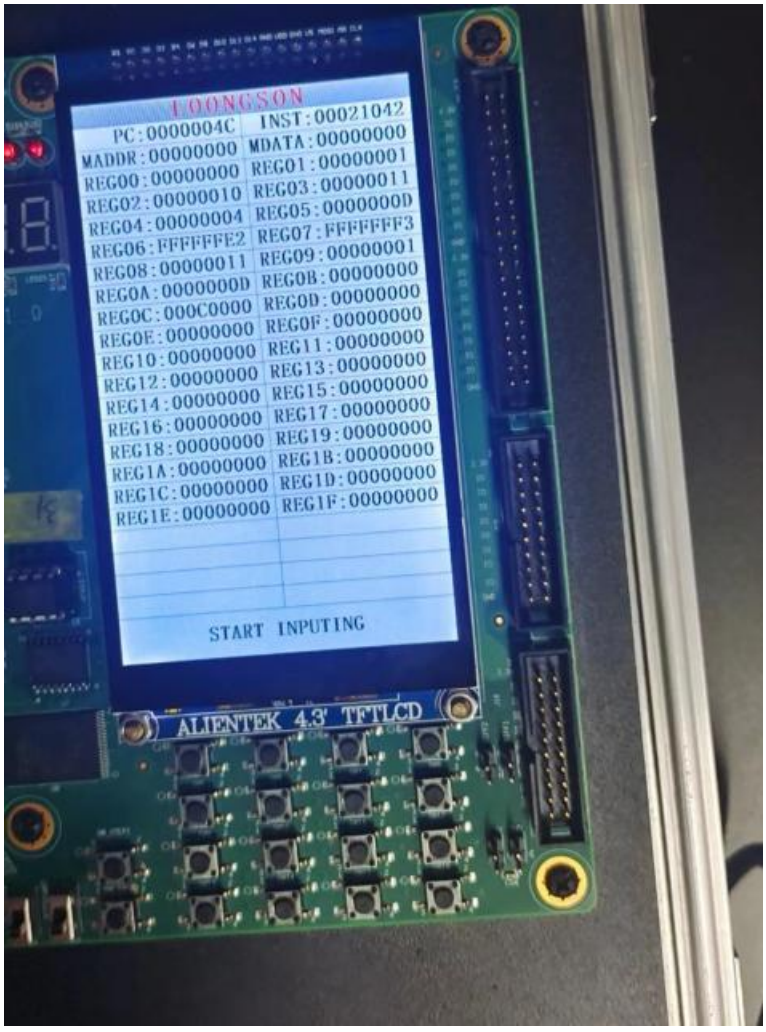
并且此时$6 的值仍为 0，尚未写入结果，这也与预期一致。



执行到最后一条指令 2C 时，**所有寄存器结果也与预期一致，这就验证了源码的实验结果了。**

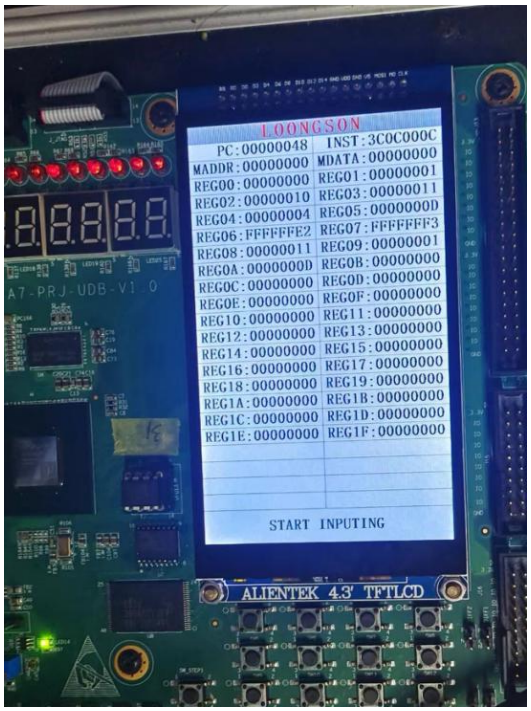
2. 对修改后的新指令进行验证：（第二次实验）

（1）异或指令执行前：



可以看到此时 INST 处成功显示了所设计的异或指令对应的十六进制数,并且
此时在执行指令前,对应写入 I 型异或指令的 REG1Eh 和写入 R 型指令的 REG1F 位置仍为 0。

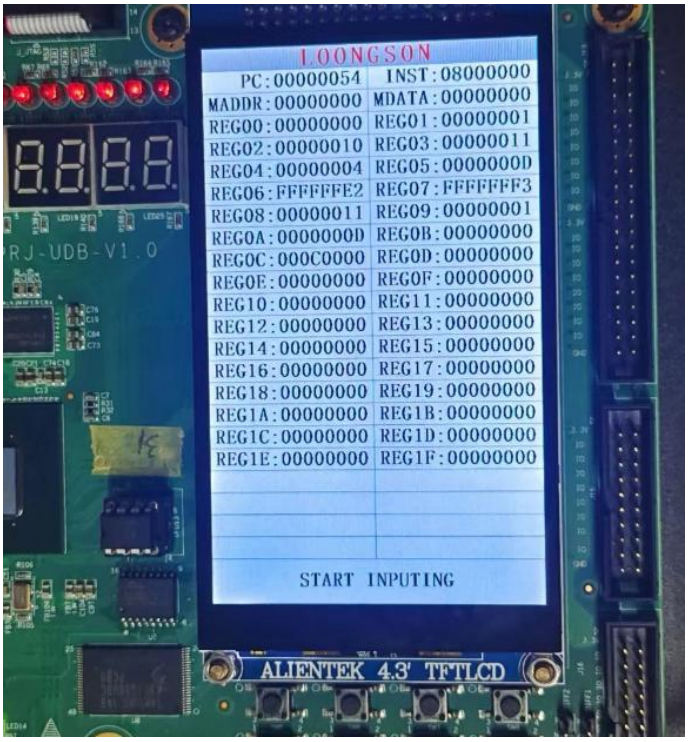
（2）异或指令执行后/大于置位指令执行前：



可以看到此时 REG1E 处成功即$30 寄存器的值被成功写入，写入的值是 REG0 即$0 寄存器的值 0 与立即数 1 异或的结果即 1，**因此验证成功，成功实现了 I 型指令异或的功能。此时也是大于置位指令执行前，可以看到此时 INST 的值成功显示为所设计的 R 型大于置位的 INST 值，而此时即将写入的 REG1F 的即$31 寄存器位置的值仍为 0。**

（3）大于置位指令执行后：


LOONGSON
PC:00000050 INST:24420008
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000008 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:000C0000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD


LOONGSON
PC:00000054 INST:08000000
MADDR:00000000 MDATA:00000000
REG00:00000000 REG01:00000001
REG02:00000010 REG03:00000011
REG04:00000004 REG05:0000000D
REG06:FFFFFFE2 REG07:FFFFFFF3
REG08:00000011 REG09:00000001
REG0A:0000000D REG0B:00000000
REG0C:000C0000 REG0D:00000000
REG0E:00000000 REG0F:00000000
REG10:00000000 REG11:00000000
REG12:00000000 REG13:00000000
REG14:00000000 REG15:00000000
REG16:00000000 REG17:00000000
REG18:00000000 REG19:00000000
REG1A:00000000 REG1B:00000000
REG1C:00000000 REG1D:00000000
REG1E:00000000 REG1F:00000000
START INPUTING
ALIENTEK 4.3' TFTLCD

可以看到此时大于置位指令成功执行，因为$3 寄存器的值 11 大于$2 寄存器的值 10，故对$31 寄存器进行置 1，可以看到此时 REG1F 的值被成功写入了 1，**因此验证成功，成功实现了 R 型指令大于置位的功能。**

6、 总结感想

1.本次实验中我最开始遇到了很多困难，比如对 MIPS 指令结构的不熟悉让我最开始回忆不起来如何进行修改，在修改过程中我也曾经忘记修改一些部分，比如对应的 alu.v 中将 alu_control 的位数调回 14 位。不过最后通过耐心地 debug，最终克服了困难，实现了改进。

2.通过本次实验，亲自将前五次所学知识和所进行对应的模块拼接起来，最终实现了实验所需的单周期 CPU。同时又根据实验要求，对 I 型指令和 R 型指令进行设计，让我在掌握了单周期 CPU 的原理和设计的同时，更加深刻地理解了 MIPS 的指令结构，理解 MIPS 指令集中常用指令的功能和编码以及相关分类都更加印象深刻。也加深了我对组成原理知识的理解。

感谢理论课老师李涛老师与实验课指导老师董前琨老师，还有每一位助教学长学 姐，感谢这一学期的认真指导与悉心传授，我会更加进一步对计算机相关原理知识探究，并将其用在我今后的学习与工作中。